CHAPTER 4

Xeon Phi Core Microarchitecture

A processor core is the heart that determines the characteristics of a computer architecture. It is where the arithmetic and logic functions are mostly concentrated. The *instruction set architecture* (ISA) is implemented in this portion of the circuitry. Yet, in a modern-day architecture such as Intel Xeon Phi, less than 20 percent of the chip area is dedicated to the core. A survey of the development of the Intel Xeon Phi architecture will elucidate why its coprocessor core is designed the way it is.

The Intel Pentium Pro-based processors, designed around a long execution pipeline and a high degree of out-of-order instruction execution, reached a power barrier to increasing processor performance by increasing frequency. The demand for computing was still in its infancy, and the computing industry started the move toward parallel and multicore programming to feed the demand. The technical computing industry needed more compute power than provided by existing multicore architecture to continue modeling real-world problems in a wide range of fields from oil and gas exploration to biomedical engineering. As an example, in the oil and gas exploration field, high-performance computing is a competitive advantage for those who can make use of these computing resources to drill efficiently for oil recovery. Reducing the number of holes drilled in the search for oil reservoirs helps the environment as well as the bottom line. However, the simulation capabilities needed for such applications require processor performance and efficiency levels to push the power envelope of current serial or multicore-based architectures. Meeting the technical computing demands of the near future will require the implementation of disruptive technologies that will make each core substantially faster.

As a result, the computing industry started looking for alternate approaches—including attached-coprocessor solutions from Clearspeed, various *field-programmable gate array* (FPGA) solutions, and graphics chips such as from AMD and NVIDIA—to improve the performance of the scientific applications. Developing and maintaining such software seemed, however, prohibitively costly for practical industrial development environments. What was needed was a universal coprocessor programming language and processor architecture to leverage the development and tuning expertise of today's software engineers to achieve the power and performance goals of future computational needs. The architecture team within Intel found that the cores based on Intel Pentium designs could be extremely power-efficient on current semiconductor process architecture thanks to short pipelines and low-frequency operations. These cores could also retain many of the existing programming models that most of the developers in the world were already using. For the technical computing industry, the value of the years of investment that have been put into software development must be preserved. The compatibility and portability of existing applications are critical criteria in selecting among different hardware platforms for technical computing applications.

Intel Xeon Phi Cores

The decision to use Intel Pentium cores started the effort to develop Intel's first publicly available *many-integrated-core* (MIC) architecture. In this architecture, a single in-order core is replicated up to 61 times in Intel Xeon Phi design and placed in a high-performance bidirectional ring network with fully coherent L2 caches. Each of the cores supports four hyperthreads to keep the core's computing process busy by pulling in data to hide latency. The cores are also designed to run at *turbo modes*—that is, if the power envelop allows, the core frequency can be increased to increase performance.

These cores have dual-issue pipelines with Intel 64 instruction support and 16 floating-point (32-bit) wide SIMD units with FMA support that can work on 16 single-precision or 8 double-precision data with a single instruction. The instructions can be pipelined at a throughput rate of one vector instructions per cycle. The core contains a 32-kB 8-way set-associative L1 data and instruction cache. There are 512 kB per core L2 cache shared among four threads, and there is a hardware prefetcher to prefetch cache data. The L2 caches between the cores are fully coherent.

The core is a *2-wide processor*—meaning it can execute two instructions per cycle, one on the U-pipe and the other on the V-pipe. It also contains an x87 unit to perform floating-point instructions when needed.

The Intel Xeon Phi core has implemented a 512-bit Vector ISA that can execute 16 single-precision floating-point or 32-bit integer and 8 double-precision floating-point or 64-bit integer vector instructions. Vector units consist of 32x 512-bit vector registers and 8 mask registers to allow predicated execution on the vector elements. Support for Scatter/Gather vector memory instructions makes assembly code generation easier for assembly coders or compiler engineers. Floating-point operations are IEEE 745 2008-compliant. Intel Xeon Phi architecture supports single-precision transcendental instructions for exp, log, recip, and sqrt functions in hardware.

The vector unit communicates with the core and executes vector instructions allocated in the U or V pipeline. The V-pipe executes a subset of the instructions and is governed by instruction-pairing rules, which are important to account for in getting optimum processor performance.

■ **Calculating the Theoretical Performance of Intel Xeon Phi Cores** For an instantiation of the Intel Xeon Phi coprocessor with 60 usable cores, running at 1.1 GHz, you can compute its theoretical performance for single-precision operations as follows:

- GFLOP/sec = 16 (SP SIMD Lane) × 2 (FMA) × 1.1 (GHz) × 60 (# cores) = 2112 for single-precision arithmetic
- GFLOP/sec = 8 (DP SIMD Lane) × 2 (FMA) × 1.1 (GHz) × 60 (# cores) = 1056 for double-precision arithmetic

---

■ **Note** Since the Intel Xeon Phi processor runs an OS inside, which may take up a processor core to service hardware or software requests such as interrupts, a 61-core processor may end up with 60 cores available for pure compute tasks.

Core Pipeline Stages

The core pipeline is divided into seven stages for integer instructions, plus six extra stages for the vector pipeline (Figure 4-1). Each stage including E and prior stages is speculative, since events such as a branch mispredict, data cache miss, or *translation look-aside buffer* (TLB) miss can invalidate all the work done up to this stage. Once it enters the WB stage, it is done and updates the machine's states.

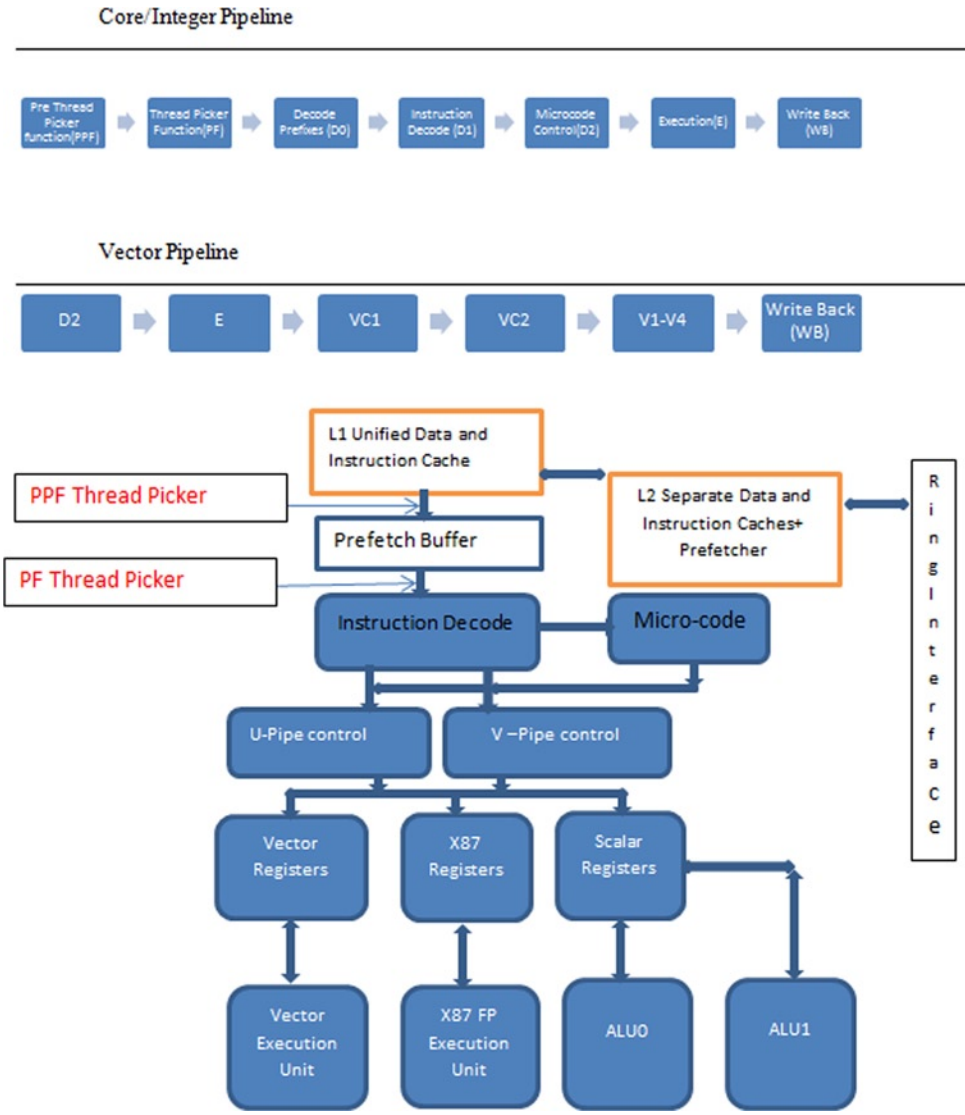

Figure 4-1. *Coprocessor core integer and vector pipeline and simplified instruction/data flow model*

Each core is *4-way multithreaded*, meaning that each core can concurrently execute instructions from four threads/processes. This helps reduce the effect of vector pipeline latency and memory access latencies, thus keeping the execution units busy.

The traditional *instruction fetch* (IF) stage is broken down into two *thread picker* stages for selecting the thread to execute: *pre-thread picker function* (PPF) and thread *picker function* (PF). The PPF stage prefetches instructions for a thread context into the prefetch buffers. There are four buffers per thread in the prefetch buffer space, and these can contain 32 bytes each per buffer. There are two streams per thread. When one of the streams is stalled, owing for example to a branch mispredict, a second stream is switched in while the branched target stream is being prefetched.

The PF stage selects the thread to execute by sending the instruction pairs to decode stages. The hardware cannot issue instructions back-to-back from the same thread in the core. To reach full execution unit utilization, at least two threads must be running at all times. This is an important caveat as it may affect execution performance in some scenarios (see Code Listing 4-1). Each of the four threads has a ready-to-run buffer (prefetch buffer) that is two instructions deep, as each core is able to issue two instructions per clock on both the U-pipe and V-pipe. The picker function (PF) examines the prefetch buffer to determine the next thread to schedule. Priority to refill (PPF) a prefetch buffer corresponding to a thread is given to the thread executing at current cycle. If the executing thread has a control transfer to a target not in the buffer, it will flush the buffer and try to load the instruction from the instruction cache. If it misses the instruction cache, a core install will happen, possibly incurring a performance penalty. The prefetch function behaves in a round-robin fashion when instructions are in the prefetch buffer. It is not possible to issue instructions from the same context in back-to-back cycles. The refill of the instruction prefetch buffer takes 4–5 cycles, which means it may take 3–4 threads running for optimal performance. When PPF and PF are properly synchronized, the core can execute in full speed even with two hardware contexts. When they are not synchronized, as in the event of a cache miss, a one-clock bubble may be inserted. A possible optimization solution to avoid this performance loss is to run three or more threads in such cases.

Once the thread picker has chosen an instruction to send down the pipe to the instruction decode stages, stages D0 and D1 decode them at the rate of two per clock. The D0 stage does the fast prefix decoding, where a given set of prefixes can be decoded without penalty. Other sets of legacy prefixes may incur a two-clock penalty. The D1 stage microcode(μcode or ucode) ROM is also a source of microcode, which is muxed in with the ucodes generated by the previous decoding stage. The processor reads the general purpose register file at the D2 stage, does the address computation, and looks up the speculative data cache. The decoded instructions are sent down to the execution unit using the U and V pipelines. U is the first path taken by the first instruction in the pair; the second instruction, if pairable under pairing rules that dictate which instruction can pair up with the instructions sent down the U-pipe, is sent down the V-pipe. At this stage, integer instructions are executed in the *arithmetic logic units* (ALU)s. Once scalar integer instructions reach the *writeback* (WB) stage, they are done. There is a separate pipeline for x87 floating-point and vector instructions that starts after the core pipeline. When vector instructions reach the WB stage, the core thinks they are done, but they are not done, because the vector unit keeps working on them until they are done at the end of the vector pipeline five cycles later. At this stage they don't raise any exceptions and will get done.

Intel Xeon Phi processors have *global stall pipeline architecture*—that is, part of the pipeline will stall if one of the stages is stalled for some reason. Modern Intel Xeon architecture has queues to buffer the stalls between the front and back end.

Many changes were made to the original 32-bit P54c architecture to make it into an Intel Xeon Phi 64-bit processor. The data cache was modified to non-blocking by implementing thread-specific flush. When a thread has a cache miss, it is now possible to flush only the pipeline corresponding to that thread without blocking other threads. When the data are available for the thread that had a cache misses, it will wake up.

Cache and TLB Structure

The details of the L1 instruction and data cache structure are shown in Table 4-1. The data cache allows simultaneous read and write, allowing cache line replacement to happen in a single cycle. The L1 cache consists of 8 ways set-associative 32-kB L1 instruction and 32-kB L1 data cache. The L1 cache access time is approximately 3 cycles, as measured in the "Understanding Intel Xeon Phi Cache Performance" section of this chapter. L2 cache is 8-way set-associative and 512 kB in size. The cache is *unified*—that is, it caches both data and instructions. The L2 cache latency could be as small as 14-15 cycles.

Table 4-1. *Intel Xeon Phi L1 I/D Cache Configuration*

Size	32 kB
Associativity	8-way
Line size	64 bytes
Bank size	8 bytes
Outstanding misses	8
Data return	Out of order

L1 data TLB supports three page sizes: 4 kB, 64 kB, and 2 MB (Table 4-2). It also has a L2 TLB that acts as a true second-level TLB for 2-MB pages or acts as a cache for *page directory entries* (PDEs) for 4-kB and 64-kB pages. If one misses L1 and also misses L2 TLB, one has to walk four levels of page table, which is pretty expensive. For 4-kB/64-kB pages, if one misses the L1 TLB but hits the L2 TLB, it will provide the PDE (Figure 4-2) directly and be done with the page translation.

Table 4-2. *Intel Xeon Phi TLB Configuration*

	Page Size	Entries	Associativity
L1 Data TLB	4 KB	64	4-way
	64 KB	32	4-way
	2 MB	8	4-way
L1 Instruction TLB	4 KB	64	4-way
L2 TLB	4 KB, 64 KB, 2 MB	64	4-way

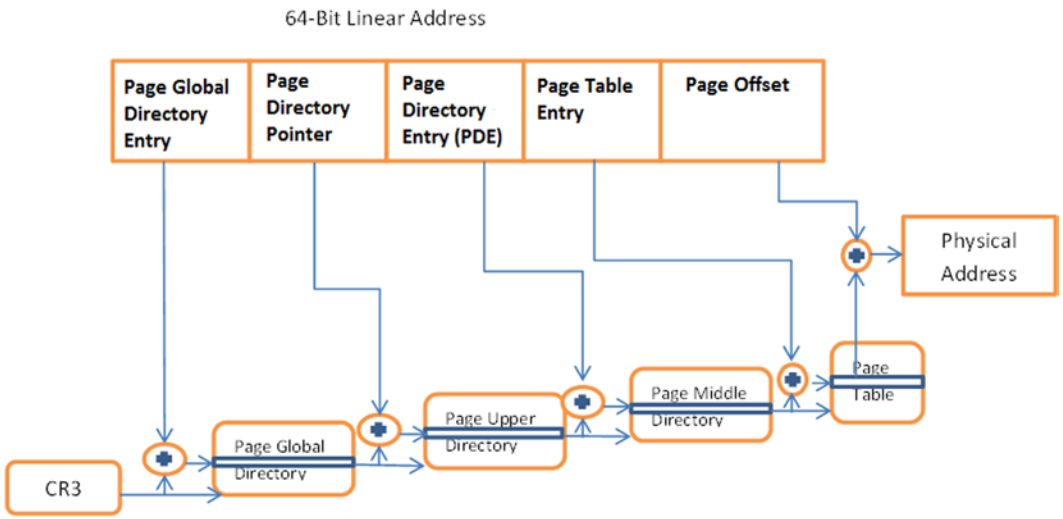

Figure 4-2. *Linear-to-physical address translation in Intel Xeon Phi coprocessor*

The page-translation mechanism allows the applications to use much larger address spaces than physically available in the processor. The size of a physical address is an implementation specific to the hardware. Intel Xeon Phi supports a 40-bit physical address in a 64-bit mode—that is, the coprocessor will generate a 40-bit physical address signal on the memory address bus of the coprocessor. Although Intel Xeon Phi supports various virtual address modes, such as 32-bit and physical address extension (36-bit) modes, I will focus here mainly on the 64-bit mode, as that is what is implemented through the micro OS running on the coprocessor. In the 64-bit mode, there is architectural support for applications to use a 64-bit linear address. The *paging* mechanism implemented in the operating system allows a linear address used by an application to map to a physical address, which can be less than 64 bits. A 64-bit linear address is used to address code, data, and stack. The micro OS running on the coprocessor uses 4-level hierarchical paging. Linear addresses generated by an application are grouped in fixed-length intervals known as *pages*. The micro OS running on Intel Xeon Phi supports 4kB and 2MB page sizes; 64KB implemented in the hardware is not yet supported by micro OS at time of writing. The application or OS may chose to move between various page sizes to reduce the TLB misses. The current micro OS implements *transparent huge page* (THP) to automatically promote or demote page sizes during an application run. You can use the `madvise` system API to control THP behavior.

The operating system creates data structures known as *page table data structures*, which the hardware uses to translate linear addresses into physical addresses. These page table data structures reside in the memory and are created and managed by the operating system or, in the case of Intel Xeon Phi, micro OS. There are four levels of page table data structures:

1. Page Global Directory
2. Page Upper Directory
3. Page Middle Directory
4. Page Table

A special processor register, *CR3*, points to the base of the data structure Page Global Directory. Each data structure contains a set of entries that point to the next lower-level table structure. The lowest-level entries in the hierarchy point to the translated page, which, when combined with the offset from the linear address, provides the physical address that can be used to access the memory location. The translation process is shown in Figure 4-2.

The linear address can be logically divided into the following steps:

1. The Page Global Directory Offset, which combines with the base address of the Page Global Directory table found in the CR3 register to locate the Page Upper Directory table.
2. The Page Directory Pointer entry, which selects an entry from the Page Upper Directory Offset table to locate the Page Middle Directory table.
3. The Page Directory Entry, which selects an entry from the Page Middle Directory Offset table to determine the page table location in memory.
4. The Page Table Entry, which selects the physical page address by indexing into the Page Table discovered in step 3 above.
5. Page Offset, which then provides the actual physical address from the page address discovered in step 4. This address is used by the hardware to fetch the data.

The sole work of the TLB is to reduce the page-walk necessary to locate the page corresponding to steps 1 through 4 and save the page address discovered in step 4 in the TLB cache.

Applications running in the operating system have a separate set of this table structure and are switched in and out by changing the CR3 register value corresponding to the application.

L2 Cache Structure

The L2 cache is the secondary cache for the core. The L2 cache is inclusive of the L1 cache. It is all right to have a cache line in L2 only, but L1 cache lines must have a copy in L2. The L2 cache associated with each core is 512 kB in size. The cache is divided into 1024 sets and 8 ways per set with 64 bytes/1 cache line per way. The cache is divided into two logical banks. The L2 cache can deliver 64 bytes of read data to corresponding cores every two cycles and 64 bytes of write data every cycle.

Multithreading

The Intel Xeon Phi coprocessor uses time-multiplexed multithreading. The PPF thread picker prefetches instructions for a thread context into the prefetch buffers, and the PF selects what entries to read from the prefetch buffer and send them to the decoder. It uses a smart round-robin mechanism to pick only threads that have work to do and avoids threads that are inactive due to various conditions such as cache miss. For optimization, you can control thread scheduling by putting delay loops in the thread that the thread picker will not schedule in the actual hardware.

To support multithreading, all architectural states are replicated four times. Microarchitectural states of prefetch buffers, instruction pointers, segment descriptors, and exception logic are replicated four times as well. The design uses ID bits to distinguish between threads for shared structures such as *instruction TLB* (ITLB), *data TLB* (DTLB), and *branch target buffer* (BTB). All memory stalls are converted into thread-specific flushes.

Performance Considerations

Run two or more threads to keep the core completely busy. Integer operations and mask instructions are single-cycle. Most vector instructions have four-cycle latency and single-cycle throughput, so having four threads will not reveal the vector unit latency if all threads are executing vector instructions. Use four threads to hide vector unit latencies.

Address Generation Interlock

Address generation interlock (AGI) latencies are three clock cycles. When one writes to a *general-purpose register* (GPR) that is used as a base or index register, these instructions have to be spaced properly, as the address generation is done in a separate stage in the pipeline. For example, consider the following instruction sequence:

```
add rbx,4
mov rax,[rbx]
```

Here the calculation of the actual linear address from `[rbx]` is done in a separate stage before the memory fetch happens at line 2 above. In this case, hardware will insert two clock-delays between these two instructions. If running more than one thread, one may not see it as instructions from other threads can run during the dead clock-cycles for another thread.

There is also dependent load/store latency. The cores do not forward the store buffer contents, even if the data is available in the buffer to fulfill the request by the load instructions. So a load followed by a store to the same location will need the store buffer to write to the cache and then read back from the cache. In this case, the latency is four clock-cycles.

If there is a bank conflict—that is, if the U- and V-pipes try to access a memory location that resides on the same L2 cache bank—the core will introduce two clock-cycle delays for the V-pipe. When the cache line has to be replaced, it involves two clock-cycle delays. So if the code has a lot of vector instructions that miss cache, the core will have to put in a two-cycle delay for each miss to do the replacement.

Prefix Decode

There are two classes of prefixes. The fast prefixes are decoded in 0 cycles with dedicated hardware blocks and include the following prefixes: 62 for vector instructions, c4/c5 for mask instructions, *register extension* (REX) for integer instructions for 64-bit code, and 0f. All other prefixes are decoded in two-cycle latencies and include the following prefixes: 66 for selecting 16-bit operand size, 67 for address size, a lock prefix, a segment prefix, and an REP prefix for string instructions.

Pairing Rules

There are specific instructions that are pairable and thus can execute in both the U- and V-pipelines. These instructions include independent single-cycle instructions. Some dependency exceptions occur where two dependent instructions can execute in pair—such as cmp/jcc, push/push, pop/pop—by incorporating the instructions' semantic knowledge into the hardware.

For pairing to occur, instructions cannot have both displacement and immediate values, and the instructions cannot be longer than 8 bytes. 62, c4, and c5 prefixed instructions with 32-bit displacement do not pair. Microcode instructions do not pair.

Scalar *integer multiples* (imul) are long latency instructions (10 cycles) in Xeon Phi as they are executed in x87 floating-point units. One needs to shift the integer operands to FP units to do the multiply and shift the results back to the integer pipeline.

Probing the Core

This section probes the behavior of the Xeon Phi core by executing various code fragments on it. This will provide you with practical experience on what the core is capable of achieving given the core architecture described so far in this chapter.

Measuring Peak Gflops

This section examines through experiments the core microarchitecture and pipelines described in the preceding sections. Code Listing 4-1 exemplifies how to measure the core computing power of a 61-core Intel Xeon Phi, of which 60 cores are available for computation while one of the cores services the operating system. Code Listing 4-1 is written in simple native code which can run on the card using the techniques discussed in Chapter 2.

The code needs to be designed such that the computation is core-bound and not limited by the memory or cache bandwidth. To do so, the data need to fit in the processor vector registers, limited to 32 for each thread in Intel Xeon Phi cores. The code uses OpenMP parallelism to run on 240 threads on the 60 core processors (with 4 threads/core). (See Appendix A for a background on OpenMP.) To build OpenMP code, I included the <omp.h> header file, which allows me to query various OpenMP parameters.

Writing the code in OpenMP will allow you to run the code from one core to multiple cores using an environment switch and gain an experimental understanding of the compute power of the processor.

Lines 42 and 43 in Code Listing 4-1 declare two constants. The first constant, 16, indicates the number of double precisions to be used for the computation. In this architecture, the vector units can work on 8 (DP) numbers per cycle. As discussed in the "Core Pipeline Stages" section of this chapter, each of the four threads per core has a ready-to-run buffer (prefetch buffer) two instructions deep, such that each core is able to issue two instructions per clock. Having 16 DP elements allows me to put a pair of independent instructions in the fill buffer. I have also made the number of iterations ITER multiple of 240, the number of maximum threads I shall be running to balance the workload.

The function-elapsed time uses the Linux gettimeofday function to get the time elapsed for computation. Line 52 declares the three arrays that I shall be working on each in each thread. I have aligned the arrays to 64-byte cacheline boundaries so that they can be loaded efficiently by the compiler. I let the compiler know however that the data are aligned by "`pragma vector aligned`" at line number 64 and 72 of Code Listing 4-1. If you look carefully,

all threads will be writing back to array a[] in line 65 and 73. This will cause a race condition and will not be useful in real application code. However, for the sake of illustrative purposes, this effect may be ignored at this time.

In lines 62 through 66 I warm up the cache and initialize openmp threads so that these overheads are not counted during the code timing loops that happen between lines 70 through 74. For easy reference, if you look at the code fragment below, line 70 is a repetition of the omp parallel first encountered at line 62. As such, all OpenMP overhead related to thread creation is captured at line 62 and the threads are reused at line 70. The OpenMP "`pragma omp parallel for`" will divide up the loop iterations statically in this default case among the available threads set by the environment variable. The pragma vector aligned at line 72 tells the compiler that the arrays a, b, and c are all aligned to the 64-byte boundary for Intel Xeon Phi and do not need to do any special load manipulations needed for unaligned data.

At lines 78 and 79, the Gflops operations are computed. As the operations in line 73 are a fused multiply–add operation, it is in effect computing two DP FLOP operations. Since there are SIZE number of double-precision elements each operated over ITER number of times by one or multiple threads, depending on the OpenMP environment variable OMP_NUM_THREADS setting, the amount of computation is equal to 2*SIZE*ITER/1e+9 in Giga floating point operations or GFLOP. Dividing this number by duration measured by the timer routines at line 68 and 76 will allow us to compute the GFLOP/seconds which are an indication of the core performance. In addition, the Intel compiler allows us to control the code generation such that we can turn vectorization on or off.

Note that I have used special array notation supported by Intel Compiler known as *Cilk Plus notation*. Cilk Plus array notation asks the compiler to work on SIZE elements of double-precision data and asserts that there is are no array data aliasing. This helps indicate to the compiler that the elements of the arrays (a, b, c in this case) are independent, thus allowing the compiler to vectorize the code, which otherwise could be ambiguous to the compiler as to its vectorizability.

Code Listing 4-1. Source Code to Measure Peak Gigaflops on Xeon Phi

```
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <omp.h>
41
42 unsigned int const SIZE=16;
43 unsigned int const ITER=48000000;
44
45 extern double elapsedTime (void);
46
47 int main()
48 {
49     double startTime, duration;
50     int i;
51
52     __declspec(aligned(64)) double a[SIZE],b[SIZE],c[SIZE];
53
54
55     //intialize
56     for (i=0; i<SIZE;i++)
57     {
58         c[i]=b[i]=a[i]=(double)rand();
59     }
60
61 //warm up cache
62     #pragma omp parallel for
63     for(i=0; i<ITER;i++) {
64     #pragma vector aligned (a,b,c)
```

```
65     a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
66     }
67
68     startTime = elapsedTime();
69
70     #pragma omp parallel for
71     for(i=0; i<ITER;i++) {
72     #pragma vector aligned (a,b,c)
73     a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
74     }
75
76     duration = elapsedTime() - startTime;
77
78     double Gflop = 2*SIZE*ITER/1e+9;
79     double Gflops = Gflop/duration;
80
81     printf("Running %d openmp threads\n", omp_get_max_threads());
82     printf("DP GFlops = %f\n", Gflops);
83
84  return 0;
85
86 }
```

Now that you are familiar with what the code is doing, you can proceed to measure the core performance of this architecture. You need to build the code with vectorization turned off, using the -no-vec command sent to the Intel compiler. To build with the Intel compiler with vectorization turned off, use the following command line:

```
Command_prompt > icpc -O3 -mmic -opt-threads-per-core=2 -no-vec -openmp -vec-report3 dpflops.cpp
gettime.cpp -o dpflops.out
```

where:

1. icpc is the Intel C++ Compiler invocation command.
2. The source file, dpflops.cpp, contains the code described in the Code Listing 4-1, and gettime.cpp contains calls to the `gettimeofday` function to get elapsed time in seconds, as shown in the following code segment:

   ```
   #include <sys/time.h>
   extern double elapsedTime (void)
   {
     struct timeval t;
     gettimeofday(&t, 0);
     return ((double)t.tv_sec + ((double)t.tv_usec / 1000000.0));
   }
   ```

3. The `-mmic` switch dictates to the compiler to generate cross-compiled code for Intel Xeon Phi coprocessor.
4. `-opt-threads-per-core=2` switch allows the compiler code generator to schedule code generation assuming 2 threads are running in each core.
5. `-no-vec` switch asks the compiler not to vectorize the code, even if it can be vectorized.

6. `-openmp` switch allows the compiler to understand the OpenMP pragmas during the compile time and link in appropriate OpenMP libraries.

7. `-vec-report3` tells the compiler to print out detailed information about the vectorization being performed on the code as it is being compiled.

In order to see the effect of vectorization, first compile the code with the –no-vec switch and run it on a single core by setting OMP_NUM_THREADS=1.

Once the code is compiled, copy the file produces, dpflops.out, to the mic card by using the scp command as follows:

```
command_prompt-host >scp ./dpflops.out mic0:/tmp
dpflops.out              100%  19KB 18.6KB/s  00:00
command_prompt-host >
```

This will upload the ./dpflops.out to the Intel Xeon Phi card and place it on the /tmp directory on the RAM disk of the card. The operating system on the Intel Xeon Phi usually allocates some portion of the GDDR memory to be used as a RAM disk and hosts the file system on the card.

You will also need to upload the dependent openmp library, libiomp5.so, to the card by the following command:

```
command_prompt-host >scp /opt/intel/composerxe/lib/mic/libiomp5.so mic0:/tmp
```

The libiomp5.so is the dynamic library necessary for running OpenMP programs compiled with Intel compilers. The Intel Xeon Phi version of the library is available at the location /opt/intel/composerxe/lib/mic/, provided you installed the compiler at the default install path on the system where you are building the application.

Once the card is uploaded with the file, you can log on to the card using the command:

```
command_prompt-host >ssh mic0
```

To run the code, you now need to set up the environment, as shown in the Figure 4-3 screen capture. The first thing you need to do is set the `LD_LIBRARY_PATH` to `/tmp` to be able to find the runtime openmp library loaded to the /tmp directory on the Xeon Phi card. Set the number of threads to 1 by setting the environment variable OMP_NUM_THREADS=1. Also set KMP_AFFINITY=compact, so that OpenMP threads for thread id 0-3 are tied to core 1 and so on.

Figure 4-3 shows that the single-threaded non-vectorized run only provided ~0.66 GFLOPs. Yet the expected DP flops for a single-core run are 2.2 Gflops [2(FMA) x1(DP non-vectorized elements) x1.1 GHz =]. As described in the "Core Pipeline Stages" section, however, the hardware cannot issue instructions back-to-back from the same thread in the core. To reach full-execution unit utilization, at least two threads must be running at all times. So running on 2 threads (`OMP_NUM_THREADS=2`) reaches 1.45 Gflops per core. As the core still uses vector units to perform scalar arithmetic, the code for scalar arithmetic is very inefficient. For each FMA on a DP element, the vector unit has to broadcast the element to all the lanes. Operate on the vector register with a mask and then store the single element back to memory. Note that increasing threads per core does not improve performance, as the instruction can be issued every cycle for this case. Now if you extend to 240 threads utilizing all 60 cores, you can achieve 86 Gflops, as shown in Figure 4-3. Let's see whether we can get close to the 1 teraflop designed for Intel Xeon Phi by turning on vectorization.

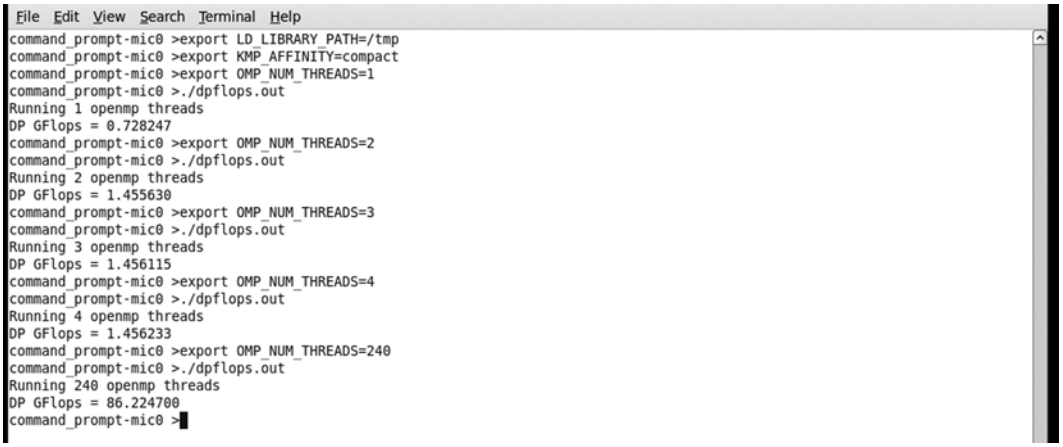

Figure 4-3. *Executing non-vectorized code on Intel Xeon Phi*

Turn on the vectorization by removing the –no-vec switch. and recompile the code with the following command line:

```
icpc -O3 -opt-threads-per-core=2 -mmic -openmp -vec-report3 dpflops.cpp gettime.cpp -o dpflops.out
```

This prints out the following compiler report:

```
dpflops.cpp(58): (col. 29) remark: loop was not vectorized: statement cannot be vectorized.
dpflops.cpp(65): (col. 16) remark: LOOP WAS VECTORIZED.
dpflops.cpp(63): (col. 9) remark: loop was not vectorized: not inner loop.
dpflops.cpp(73): (col. 16) remark: LOOP WAS VECTORIZED.
dpflops.cpp(71): (col. 4) remark: loop was not vectorized: not inner loop.
```

This report shows that Lines 65 and 73 of Code Listing 4-1 have been vectorized by the compiler. Let's upload the binaries to Intel Xeon Phi by scp command as before and run it under various conditions. With an FMA and double-precision arithmetic which can work on 8 DP elements at a time, you should see around `2 x (for Fused Multiply and Add (FMA)) x 8 (DP elements) x 1.1 GHz = 17.6 Gflops` per core.

Figure 4-4 shows the results of experimentation with the vectorized double-precision code. With 1 thread, you are able to reach 8.7 Gflops, not 17.6 as expected. As described in the "Core Pipeline Stages" section, this is due to the issue bandwidth of the core pipeline not being fully utilized by dispatching instructions only every other cycle. In order to achieve this, you need to use at least two threads. By setting OMP_NUM_THREADS=2 with KMP_AFFINITY=compact, you made sure the code was running on the same core, but two threads were executing and scheduling instructions every other cycle. This kept the execution unit fully utilized and it was able to achieve near-peak performance of 17.5 Gflops per core on the double-precision fused multiply–add arithmetic. Observe in Figure 4-4 that increasing the number of threads to 3 and 4 did not improve performance since we already saturated the double-precision execution unit on the core. Utilizing all 60 cores, with 2 threads each you could achieve 1022 Gflops, which is near the theoretical peak of 1055 Gflops on this coprocessor. You can also see that I have set KMP_AFFINITY=balanced to make sure that the threads are affinitized to different cores to distribute computations evenly when the number of threads is less than the maximum number of threads needed to saturate the cores, in this case 240 threads. Since I am running 120 threads with 2 threads per core on 60 cores, having affinity set to balanced, there will be two threads per core distributed across cores.

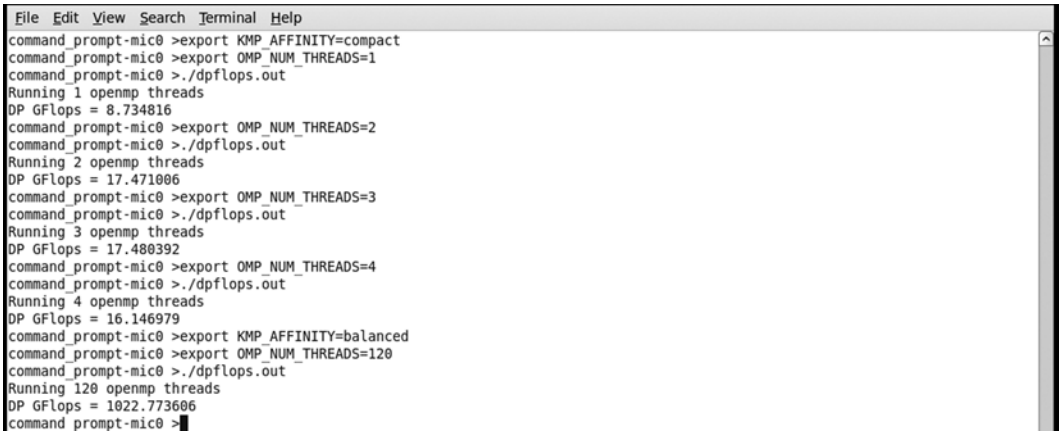

Figure 4-4. *Execution of vectorized double-precision code on Intel Xeon Phi*

Let's turn our attention to single-precision arithmetic. You would expect the performance to be 2x the double precision. The changes to the code segment in Code Listing 4-1 will be to change double to float. We need to keep the float calculation the same, as the number of operations is the same. Since the vector unit can work on 16 elements at a time with the float, in order to be able to fill up the dispatch queue for each thread, we would need twice as many elements as double, and so I have adjusted the size to 32 elements instead of 16 elements as shown in Code Listing 4-2.

Code Listing 4-2. Single-Precision Peak Gigaflops Measurement

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

unsigned int const SIZE=32;
unsigned int const ITER=48000000;

extern double elapsedTime (void);

int main()
{
    double startTime, duration;
    int i;

    __declspec(aligned(64)) float a[SIZE],b[SIZE],c[SIZE];


    //intialize
    for (i=0; i<SIZE;i++)
    {
        c[i]=b[i]=a[i]=(double)rand();
    }

```

```
61 //warm up cache
62     #pragma omp parallel for
63     for(i=0; i<ITER;i++) {
64     #pragma vector aligned (a,b,c)
65     a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
66     }
67
68     startTime = elapsedTime();
69
70     #pragma omp parallel for
71     for(i=0; i<ITER;i++) {
72     #pragma vector aligned (a,b,c)
73     a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
74     }
75
76     duration = elapsedTime() - startTime;
77
78     double Gflop = 2*SIZE*ITER/1e+9;
79     double Gflops = Gflop/duration;
80
81     printf("Running %d openmp threads\n", omp_get_max_threads());
82     printf("SP GFlops = %f\n", Gflops);
83
84  return 0;
85
86 }
```

For single-precision arithmetic, I did the same experimentation as before and the output is captured in Figure 4-5. You can see that the single-precision behavior is the same for single-core execution where it can execute optimally with 2 threads per core to utilize the execution bandwidth properly, which it cannot do with a single thread. However, adding more threads to the core on this core-bound code does not help. Binding the threads to a core was enforced by the KMP_AFFINITY switch. After changing the affinity to balanced, we allowed the threads to spread out to various cores first to make use of independent execution units. Making the number of threads 120 (2 x number of cores for this case), we were able to get the performance to 1979 Gflops.

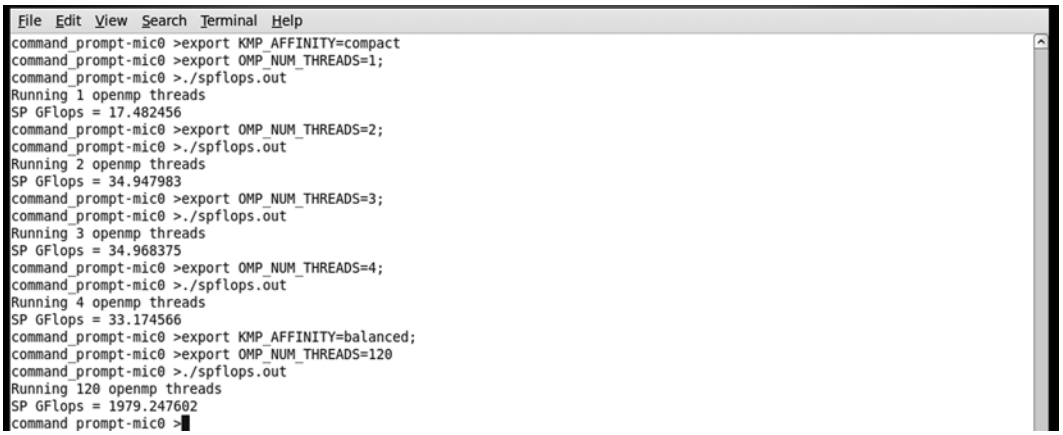

Figure 4-5. *Execution of vectorized single-precision vector code on Intel Xeon Phi*

Understanding Intel Xeon Phi Cache Performance

In order to understand the cache latencies, I shall be using the publicly available memory latency benchmark component of lmbench benchmark available at http://www.bitmover.com/lmbench/. I downloaded lmbench3 to do the experiment.

The lmbench was developed by its authors to make the benchmark portable and written in C. That was the reason I picked the benchmark to explore the memory hierarchy of Intel Xeon Phi architecture. One of the benchmark components that I was interested in was lat_mem_rd, which finds out memory-read latencies at various memory hierarchies.

In order to build this software, I downloaded the lmbench3 source from the benchmark download page. I set CC to point to icc or icpc, the Intel compiler to build the suite. I modified the scripts/compiler file as follows to point to icc:

```
/bin/sh

if [ "X$CC" != "X" ] && echo "$CC" | grep -q '`'
then
  CC=
fi

if [ X$CC = X ]
then  CC=cc
    for p in `echo $PATH | sed 's/:/ /g'`
    do   if [ -f $p/gcc ]
        then  CC="icc -mmic"
        fi
    done
fi
echo $CC
```

Once the Make file and compiler were set properly, I did a `command_prompt> make -f Makefile.mic` to build the benchmark binaries, which in my case ended up in the folder `bin/x86_64-linux-gnu`. The next step was to copy the benchmark binary I was interested in, lat_mem_rd, to the /tmp directory on the coprocessor micro OS using the scp command.

Once the benchmark binary was transferred to the card, I ran the latency benchmark as `command_prompt-mic0 >./lat_mem_rd -P 1 -N 10 32 64`.

The benchmark ran with 32 MB of memory that filled the L1 cache, in single-threaded mode with 64-byte stride, so I am reading data from different cache lines in subsequent calls. The output is captured in the screenshot in Figure 4-6. You can see that for sizes up to 32 kB, which is the L1-D cache size for each core, the cache latency is about 2.8 ns, which is approximately 3 cycles on this coprocessor. Moving beyond L1-D cache sizes, you can see that the latency goes up to 13.27 ns (~14.5 cycles).

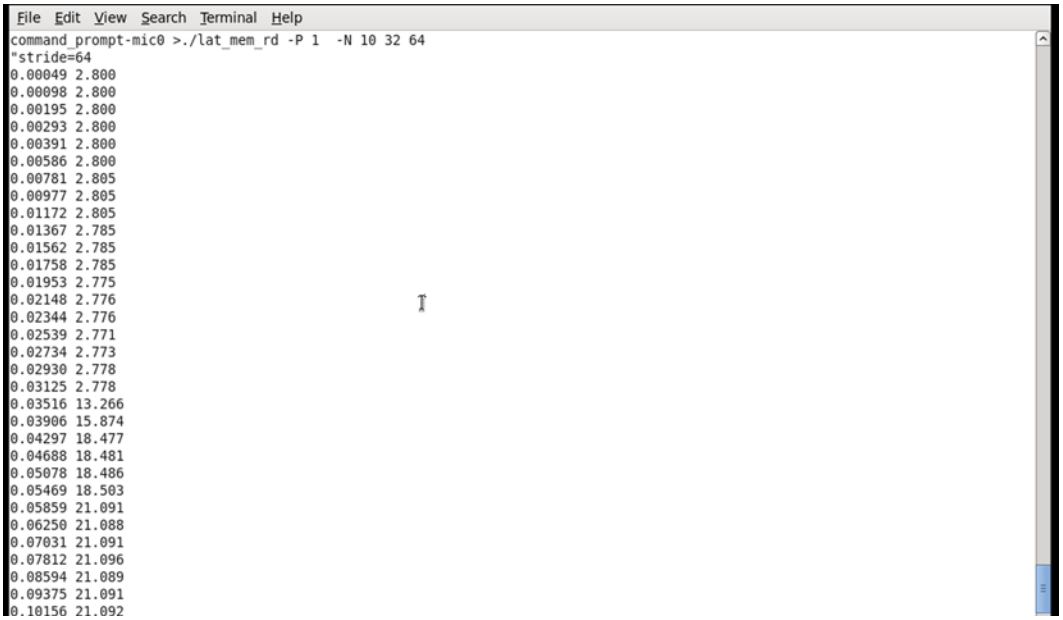

Figure 4-6. *lmbenchmark lat_mem_rd running on Intel Xeon Phi*

Summary

This chapter surveyed the Xeon Phi core architecture. You examined the core pipeline and the instruction flow through the pipeline. You learned how to compute the theoretical peak Gigaflops of the machine for single-precision and double-precision floating-point arithmetic, and how to measure the practical gigaflops achievable on the hardware by writing simple code. You looked at the cache and translation lookaside buffer (TLB) structures, which are critical to achieving high performance on the Xeon Phi coprocessor.

The next chapter will cover the cache architecture in detail. You will learn the cache protocol that you will need to maintain cache coherency and the tools you will need to measure the cache latency at each cache level.